AFRL-RI-RS-TR-2017-178

THE FRACTURE PROJECT

THE CHARLES STARK DRAPER LABORATORY, INC.

SEPTEMBER 2017

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2017-178 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN DRAGER
Work Unit Manager

/ S /
JOHN MATYJAS
Technical Advisor, Computing and Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
SEPTEMBER 2017	FINAL TECHNICAL REPORT	AUG 2012 – MAY 2017

4. TITLE AND SUBTITLE

THE FRACTURE PROJECT

5a. CONTRACT NUMBER
FA8750-12-C-0261

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Chris Casinghino

5d. PROJECT NUMBER
HACM

5e. TASK NUMBER
RE

5f. WORK UNIT NUMBER
DT

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Charles Stark Draper Laboratory, Inc.
555 Technology Square
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2017-178

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2017-4398
Date Cleared: 12 SEP 2017

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes research and testing carried out as "the voice of the offense" in the HACMS program, ensuring that the development and verification tasks undertaken by the Blue Teams focused on preventing realistic attacks on the demonstration systems. To this end, this effort (a) conducted extensive penetration testing of original and secured demonstration platforms and (b) developed novel formal methods-based tools to directly analyze software produced by Blue Team performers. This report describes how this approach resulted in the detection of numerous vulnerabilities over the course of the program and explains the research contributions made by the formal methods team in the development of a collection of new static analysis tools.

15. SUBJECT TERMS

Penetration Testing, Formal Verification, Red Team, High Assurance Cyber Military Systems

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**STEVEN DRAGER**
U	U	U	UU	45	19b. TELEPHONE NUMBER (*Include area code*) **NA**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. SUMMARY

This report describes research and testing carried out by The Charles Stark Draper Laboratory (Draper) and its subcontractors Assured Information Security (AIS) and Oxford University (Oxford) as the Red Team in the Defense Advanced Research Projects Agency (DARPA) High-Assurance Cyber Military Systems (HACMS) program under contract FA8750-12-C-0261.

The Draper team acted as "the voice of the offense" in the HACMS program, ensuring that the development and verification tasks undertaken by the Blue Teams focused on preventing realistic attacks on the demonstration systems. To this end, we (a) conducted extensive penetration testing of original and secured demonstration platforms and (b) developed novel formal methods-based tools to directly analyze software produced by Blue Team performers. This report describes how this approach resulted in the detection of numerous vulnerabilities over the course of the program and explains the research contributions made by our formal methods team in the development of a collection of new static analysis tools.

Over the course of the program, we applied our penetration testing technique described in Section 2 to initial vulnerability assessments on the unmodified platforms, and additional assessments on the secured platforms at the end of each phase. The initial assessment of each vehicle platform found numerous vulnerabilities that were made available to the Blue Teams to focus their formal verification efforts. While the final assessment demonstrated that the Blue Teams succeeded in building systems with an unprecedented level of security against many important classes of attack, vulnerabilities in these systems remained at the end of the program.

Our formal methods analyses focused on Blue Team code implemented in Galois' Ivory/Tower language. Here, we obtained a formal proof that the LED blink system behaves correctly. We also demonstrated that we could find a manually discovered replay attack using our Failures Divergence Refinement (FDR) tool, described in Section 2.2.1; and by analyzing a model extracted from the generated C code with CspGen (Section 2.2.3).

2. INTRODUCTION

The Draper Team, composed of the Charles Stark Draper Laboratory, Assured Information Security, and Oxford University, was funded to provide the Government with an integrated solution for Red Teaming and Penetration Testing for the DARPA HACMS program. Our technical approach developed new and innovative methods for reliably discovering vulnerabilities in increasingly secure systems. Our team consists of:

- Draper, a premier not-for-profit controls and systems engineering company in the U.S., with core capabilities in red teaming, systems engineering, autonomous vehicle integration, and formal methods
- AIS, one of the leading penetration testing companies in the United States, regularly conducting classified penetration tests for the Federal Government and unclassified commercial work
- Oxford University—with over twenty years of formal methods experience—provides the team with the world's leading expert in applying formal methods testing to military systems

The HACMS goal was to create technology to build *high-assurance* cyber-physical systems, where high-assurance means ***functionally correct, safe, and secure***. HACMS does this using a clean-slate formal methods-based approach. Our Red Team, Technical Area 5 (TA5), focuses on assessing the security of the targeted systems. Our technical approach assesses security using proven penetration testing techniques and our novel formal methods tools.

Our approach had two tightly-coupled components, bridged by Draper's technical and management expertise: state-of-the art penetration testing techniques and a new formal methods toolchain.

2.1 Penetration Testing

The first component is penetration testing performed by AIS, using their demonstrated state-of-the-art techniques. Penetration testing evaluates the security of an embedded system by simulating an attack from malicious outsiders without authorized access. This process involves active system analysis for potential vulnerabilities caused by poor or improper system configuration, known and unknown software flaws, or operational weaknesses.

Over the course of the program, the Red Team applied this penetration testing technique in initial vulnerability assessments on the unmodified platforms, and additional assessments on the secured platforms at the end of each phase. The initial assessment of each vehicle platform found numerous vulnerabilities. This information was made available to the Blue Teams, who used it to focus their formal verification efforts. Some of these vulnerabilities were considered out of scope for the objectives of the HACMS program. Via the intermediate end-of-phase assessments and regular collaboration with the Red Team, many additional vulnerabilities were identified and eliminated from the final delivered systems.

A more detailed description of the penetration testing strategy employed can be found in Section 3.2. The results of our penetration testing are catalogued in the end-of-phase Vehicle Security Assessment Reports and in the Final Vehicle Security Assessment Report. Therefore, this Final

Report describes these results at a high level in Section 4.1, but refers to the Vehicle Security Assessment Reports for the detailed conclusions.

2.2 Formal Methods Tools and Analysis

The second component uses formal methods to discover vulnerabilities in embedded systems. This approach is based on two insights about the nature of the problem posed by HACMS. First, security has increasing significance at higher levels of abstraction. Second, software vulnerabilities and flaws manifest themselves in the physical implementation and machine code. We have developed technology to extract formal models from the source code of as-built systems, allowing us to perform a formal, independent verification and validation of the implementation's security.

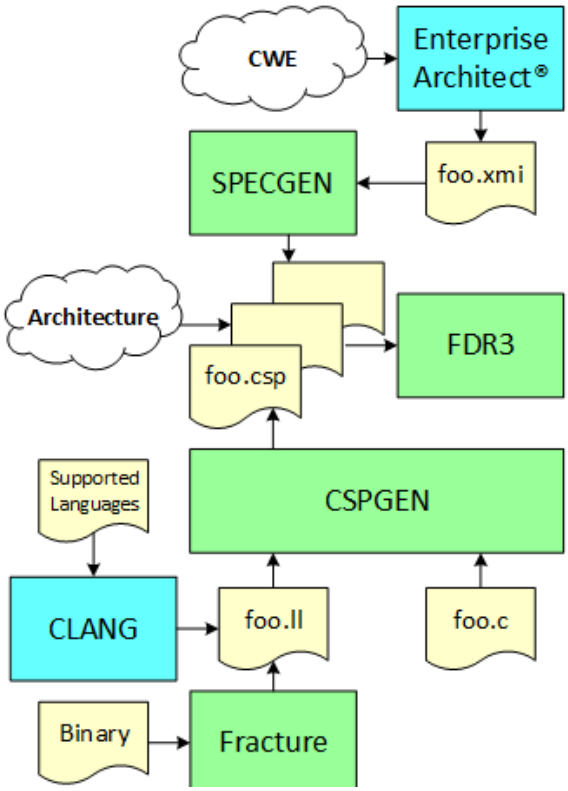

Figure 1: Formal Toolchain Architecture

Figure 1: Formal Toolchain Architecture illustrates the formal tool architecture. Tools developed by the Draper team are depicted in green boxes, while blue boxes depict external tools that are compatible with our approach. The Draper formal assessment tools are centered around the Communicating Sequential Processes (CSP) language. Failures Divergence Refinement, the Oxford model checker, can be used to formally verify properties of CSP programs. Section 3.3.1 provides more background on CSP and the application of FDR to formal verification tasks. The other tools shown in Figure 1: Formal Toolchain Architecture are used to translate systems to CSP.

This section briefly describes the purpose of each tool. Descriptions of the research conducted in the development of the tools may be found in Sections 4.2 - 4.5. Over the course of the HACMS program, these tools were used in several analyses of software developed using the Galois developed Ivory language and its Tower concurrency library. These engagements are described in Section 4.6.

2.2.1 FDR

FDR is the Oxford model checker for CSP and has been under continuous development since the 1990s. It is used to check properties of the models constructed by the other pieces of our tool-chain. Over the course of the program, FDR has undergone a complete rewrite, its model-checking speed has been improved by an order of magnitude, and substantial features have been added including the addition of type-checking for CSP and support for parallel processing of model checking. This research is described in Section 4.2.

2.2.2 SpecGen

Draper has developed a graphical capture capability for hierarchical and concurrent state machines, a subset of statecharts. This tool, SpecGen, translates statecharts drawn in the commercial Enterprise Architect (EA) modeling tool to CSP. These CSP translations may be analyzed directly to verify properties of a system model, or may be used as a specification to compare with a CSP model extracted from a source code implementation. This research is described in Section 4.3.

2.2.3 CspGen

CspGen is a Draper-developed tool that builds CSP models of programs written in imperative languages, like C. These models may be analyzed using the FDR3 model checker to check properties of the initial program. Draper uses this tool in the analysis of software provided by the other performers. The research undertaken in CspGen's development is described in Section 4.4.

2.2.4 Fracture

Fracture is a decompiler that translates ARM binaries to the Low Level Virtual Machine (LLVM) intermediate language. Draper implemented Fracture as a modification and extension of the LLVM compiler suite and has made it publicly available. Fracture's novel decompilation strategy is described in Section 4.5.

2.2.5 Formal Assessments

Application of the formal tools described above to Blue Team code focused on systems developed in Galois' Ivory language and its Tower concurrency library. This research direction was chosen, following the advice of the DARPA HACMS Program Manager, because Ivory/Tower compiles to C, a language understood by our tools, and because these programs have an understandable and formalizable specification. There were two main thrusts of this analysis: (1) verifying as-implemented Ivory/Tower programs, and (2) verifying the Tower concurrency model directly. This research, which resulted in formal verification of key correctness and security properties for Ivory/Tower software, is described in greater detail in Section 4.6.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

Our approach applied two tightly-coupled verification components. The first is penetration testing performed by AIS using their demonstrated state-of-the-art techniques. The second uses formal methods to automatically analyze the behavior of systems. We begin by describing the high-level assessment procedure undertaken by our team, and then describe the methods and assumptions of the two techniques in more detail.

3.1 Overall Procedure

The HACMS Red Team acted as "the voice of the offense" to ensure the development and verification tasks undertaken by the Blue Teams focused on preventing realistic attacks on the demonstration systems. At the beginning of the program (and again at the beginning of Phase 3, when a new platform was selected), an Initial Assessment of each platform was performed. As expected, substantial vulnerabilities were discovered in every platform, and these results were documented in Initial Vehicle Assessment Reports and provided to the Blue Team performers. Of these vulnerabilities, some were determined to be out-of-scope for the HACMS program objectives, and the rest guided the Blue Team work on securing the platforms.

As the teams delivered platforms at the end of each phase, the modified systems were assessed. The vulnerabilities were documented in end-of-phase vehicle assessment reports and provided to the Blue Teams for work in the next phase. Simultaneously, throughout the phases, the Red Team developed novel static analysis verification tools and worked with performers to apply these tools to their systems. Application of the developed formal static analysis tools centered around an examination of the Galois Ivory programming language and its Tower concurrency system. Verification of these systems involved a combination of hand-written, high-level CSP versions of their intended model of computation and extracted CSP models of the as-implemented systems.

3.2 Penetration Testing

Penetration testing was carried out primarily by AIS, a member of the Draper team. This section outlines the state-of-the-art penetration testing methodology employed by the AIS team.

The AIS vulnerability assessment methodology is a cyclic assessment process where the team's knowledge of a target evolves, and new test cases and attack vectors are identified and incorporated into the assessment. The testing team familiarizes themselves with the basic functionality of the system through standard user interaction and analysis of the functional system specification. The team then identifies potential areas of weakness in the target system's design and implementation. Further experience with the target while analyzing these potential weaknesses provides the team with further system knowledge and identifies additional potential attack vectors. This cyclic process repeats throughout the testing process, making the team extremely familiar with the target.

The approach is used to identify security vulnerabilities in software systems, computer networks, infrastructure devices, wireless networking equipment, and embedded systems. The basic process taken by AIS to perform a security analysis of any software system or hardware device is guaranteed to follow the same basic steps every time. The roots of this process lie in the fundamental engineering need to fully understand and evaluate any piece of technology that is tested. This process is separated into five major phases:

- Target Understanding
    - Target Familiarization
    - Behavioral Observation
- Design Review
- Disassembly & Reverse Engineering
- Target Analysis
- Vulnerability Assessment
    - Vulnerability Identification
    - Vulnerability Testing

The first phase of target analysis provides the general level of understanding required to effectively analyze any system. Marketing material, design documentation, specifications, user manuals, and administration guides are all analyzed. Reviewing this material provides a thorough understanding of the system, its goals and functionality, supporting components, and the anticipated concept of operations (CONOPS).

Using the knowledge learned in the document review, some basic tests are performed against the target. The team interacts with and monitors the system to identify basic functional characteristics and behavioral traits as it performs its normal tasks. This process allows the Red Team to develop an intimate knowledge of the system while becoming familiar with standard user level interaction, as well as a better understanding of procedures to configure, administer, and operate the system. This helps to identify how individual components interact with each other. The knowledge learned in this phase furthers understanding of the target system's design and often identifies potential areas of interest for later security analysis.

An understanding of the basic system's features and components allows a reference model to be designed and built to identify additional potential weaknesses. When building the reference model, the AIS analysis team incorporates assumptions about resource and time constraints the system builders may have faced into the reference model's design. The Red Team may make assumptions about the problem types encountered during the development process and concessions designers may have made to meet requirements or deadlines. These assumptions are based on our own experience with designing and developing systems, as well as other systems we have evaluated in the past. This step's goal is identifying the potential areas within the original system where designers and developers may have encountered problems or limitations. Based on the information generated during this phase, assumptions can be made about where vulnerabilities may exist within the system and where to focus initial analysis efforts.

Using these potential weak areas as a starting point, the team begins to disassemble and reverse engineer the software and hardware. This process exposes the system's intricate configuration details, its underlying application structure, hardware and software properties, and component interaction that may not normally be available. Analyzing this data provides the system's low-level details and allows the team to generate a complete and thorough definition of the system's functionality, behavior, and potential weaknesses. Using this material and our increased system understanding, individual subsystems are identified and an overall block diagram of the system and its functional components are developed.

Directly probing the individual components identified within the system representation assists in

the overall target analysis process. Interacting with the individual system components provides a mechanism to identify where vulnerabilities may exist and which system components (both major and minor) are most vulnerable. The general process followed and the actions performed are normally very similar across different systems, although interactions with any individual component may be target specific. To properly analyze a target and identify the weaknesses between components we observe the system while it performs its standard actions, collecting and analyzing the data the system generates and exchanges. We often collect this data using software mechanisms (*e.g.*, a debugger) or hardware components (*e.g.*, a serial or Joint Test Action Group (JTAG) interface). These mechanisms monitor data flows between system applications and components, and provide the system's behavioral and functional details required to further understand and investigate the system.

Using the information from the previous steps, the vulnerability identification process tries to demonstrate observable impacts on the target. Each interest point we have defined is thoroughly investigated and the system's behavior is monitored and documented for each test case. Some of the attack vectors or exploitation techniques tested in this process may be like those previously encountered in other tests or presented by vulnerability researchers. However, we expect most of the attack vectors we pursue will be unique and driven completely by the information gained during analysis. The vulnerability identification process focuses primarily on identifying situations where:

- Input is provided or passed, but is not properly validated, leading to code execution
- System output is not properly controlled/secured, allowing information leakage and data exfiltration
- Access to critical system components is not controlled using proper authentication or authorization mechanisms allowing adversarial access
- Communication protocols are not authenticated or encrypted, allowing attackers to monitor, manipulate, or inject network communications
- Software flaws allowing resource exhaustion or system crashes induced by internal or external inputs

Identifying a potentially vulnerable system component drives development of a tailored vulnerability test case. This process and the tools used to generate and execute these individual test cases are system and component specific and vary across test environments and targets. The goal of these test cases is to demonstrate the vulnerability. We do this by providing data or inputs to the system or component with the goal of having a negative impact on the system's execution, integrity, or availability. This vulnerability test is monitored and observed using the same processes from the earlier phases. Test case refinement and adaptation, as well as expanding to cover other focus areas, follow the same monitoring and observation process.

3.3 Formal Verification

Formal tool development was carried out primarily by Draper and Oxford. The Draper tools are focused around the Communicating Sequential Processes language as a "lingua franca". Draper built tools that translate system models and source-code implementations to CSP. Simultaneously, Oxford enhanced FDR, the CSP model checker. Using FDR, the extracted models of HACMS

systems can be explored and checked for vulnerabilities. In the remainder of this section, we describe CSP and outline our approach to modeling and evaluating systems with it.

3.3.1 The CSP Language and Refinement Checking

The CSP language [1, 2, 3] is a *process algebra*. Originally invented by Tony Hoare in 1978 [4], CSP has seen continuous development and use in academia and industry as a model of concurrent systems since its introduction [5]. CSP programs, also called *processes*, can intuitively be thought of as collections of concurrently running threads that communicate with each other and with the external world via *events*. We refer the reader to the FDR tutorial [6] or one of the books cited above for a complete introduction to the language. This report will focus on a few key features and a description of CSP's use in analysis of systems.

When writing a CSP program, one picks an *alphabet* that codifies what events can occur. For example, when using CSP to build models of a C program, the alphabet of events might contain "read" and "write" memory operations. When considering a more abstract system, like a model of the classic "dining philosophers" concurrency example [1], the alphabet would contain events that are correspondingly more abstract (like events representing a philosopher sitting, or picking up a fork, or eating). This generic notion of events allows the use of CSP to model a wide variety of systems at different levels of granularity.

CSP is a useful language for formal analysis of systems because it has a formal semantics that support a natural notion of *refinement*. In general, questions about CSP models are phrased in terms of a high-level, relatively abstract process `S` representing a *specification* for the system, and a low-level, relatively detailed process `I` representing an *implementation* of the system. We say that `I` *refines* `S` when every possible behavior of `I` is also a behavior of `S`. This notion of refinement can be used to capture nearly any relevant safety, security or correctness property of a cyber-physical system. The primary purpose of FDR, the Oxford CSP model checker we employed, is to check refinement between two processes in an extremely efficient and parallelizable manner.

3.3.2 Modeling Cyber-Physical Systems with CSP

Draper follows an iterative, four step process when modeling and verifying a system via CSP refinement checking.

Step 1: Build a Model of the System's Environment.

The first step is to identify and model the relevant environment of the system being verified. For example, if the system under test is an HVAC controller, it may expect to interact with its environment by turning on and off the air conditioning. A formal model of this environment will codify its constraints, like the idea that the AC can only be "turned on" when it is in the "off" state, and vice versa.

As another example, consider the verification of a computer program written in C. This program will expect to interact with a persistent memory by reading and writing to particular addresses. The program may also use functions from a binary library, or make system calls that depend on its operating system. If we wish to model the behavior of this program in CSP, we need to model the effects of these interactions.

Building a model of this environment begins by adding events representing external interactions to the alphabet of the system, as described in Section 3.3.1. Then CSP programs that model the environment and the effects of these events are written, typically by hand. It is important that these models accurately capture the real environment of the system in question at the appropriate level of detail. This can often be achieved by careful adherence to relevant documentation. For example, when building an environment for C programs, the Draper team worked directly from the C language definition [7] and tested the resulting models to ensure they behaved as expected.

Step 2: Identify and Codify System Specifications.

Next, we identify and formally state the safety, security, or correctness properties that we'd like to verify for the relevant system. These properties take the form of high-level "specification" CSP programs, as described in Section 3.3.1. They can come from many sources, like government requirements, coding standards, or discussions with the system implementers.

In some cases, these will be standard properties with fixed definitions: for example, one common property is "deadlock freedom", ensuring the system never enters a "stuck" state. In other cases, they can be properties that are specific to the system in question: for example, in Section 4.3.1 we show how to capture the property "after sitting, no philosopher stands without eating" for a model of the "dining philosophers" problem, and in Section 4.6.1 we show how we captured the property that two LEDs should blink on and off indefinitely for a model of an Ivory/Tower program.

These specifications often refer to the environment model defined in Step 1. For example, in the case of the blinking LEDs, events that represent a light turning on or off form part of this environment.

Step 3: Build or Extract an Implementation Model.

The previous step resulted in CSP processes representing the system's specification. In this step, we obtain a process representing its implementation.

In the HACMS program, we built CspGen, a tool to automatically extract such processes from programs written in the C or any language that can be compiled to the LLVM Intermediate Representation (IR) with the Clang compiler. This results in a very detailed model of the system's behavior, and ensures that our verification applies to the actual behavior of the as-built system. The research undertaken in the design and implementation of CspGen is described in Section 4.4.

Like the specifications, these implementations typically refer to the environment model built in Step 1.

Step 4: Perform Verification

At this stage, we have a formal definition of the relevant safety, security or correctness properties, and a formal model of the system under test and its environment. The last step is to apply the FDR refinement checker to see if the implementation meets its specification.

Often, an attempt to perform Step 4 results in a need to iterate upon the previous steps. This can occur for many reasons. For example, it may be that the environment model did not capture some real-world constraint that the implementation relies on, and a more precise version can be created. Alternatively, it may be the case that the implementation model is too complex for the refinement checker to handle in the available time. In this case, several strategies are available, like improving

CspGen to generate more efficient models, or decomposing the system into components that can be checked individually and combined at a higher level.

Examples of verification of HACMS software are described in Section 4.6.

4. RESULTS AND DISCUSSION

This section provides the results of the activities described at a high level in Section 3, and describes, in detail, the research that went into the development of the tools the Draper team has delivered to the Government. We begin in Section 4.1 with a high-level description of the results of our penetration testing, with additional detail provided in the Final Vehicle Security Assessment Report. In Sections 4.2 - 4.5, we describe the research conducted during development of the static analysis toolchain outlined above in Section 2.2. In Section 4.6, we describe the application of that toolchain to software developed in Galois' Ivory/Tower system. Finally, Section 4.7 catalogs the relevant academic papers published by Red Team members.

4.1 Results of Penetration Testing

Over the course of the program, the Red Team conducted initial vulnerability assessments on the unmodified platforms, and additional assessments on the secured platforms at the end of each phase. These results have been delivered in individual reports throughout the program - we do not repeat all of these findings here, but summarize the most important points.

The initial assessment of each vehicle platform found numerous vulnerabilities. This information was made available to the Blue Teams, who used it to focus their formal verification efforts. Some of these vulnerabilities were considered out of scope for the HACMS program. Via the intermediate end-of-phase assessments and regular collaboration with the Red Team, many additional vulnerabilities were identified and eliminated from the final delivered systems.

The final assessment demonstrates that the Blue Teams succeeded in building systems with an unprecedented level of security against many important classes of attack. This assessment also illustrates that vulnerabilities in these systems remain. The remaining issues with the systems fall into several categories:

- Specification weaknesses: In some cases, the specifications used in the design of the systems did not match the intuitive desired properties. This is illustrated by a geofencing violation on an air platform, and by a technique for causing a ground platform to crash into obstacles because of a poor model of its deceleration capability.

- Communication weaknesses: Communications security is a complex area of system development. Early in the program, a decision was made that many aspects of communications security were out of scope for the HACMS program. As an unsurprising result, the final systems exhibit communications vulnerabilities.

- Toolchain misuse: As illustrated in the final assessment report, the intended and verified uses for each tool are not always clear to non-expert users. As a result, these users may believe they are getting more security guarantees than are actually available, resulting in an insecure system.

These findings do not call into question the revolutionary advances in security and resilience made by the HACMS performers. However, they do suggest that work remains in learning how to apply these technologies throughout an entire system and to help non-experts check that the formal guarantees accurately capture the desired security properties.

4.2 FDR Research and Development

FDR, the Oxford CSP refinement checker, is the primary model checking back-end for our tools. Its place in our toolchain is illustrated in Figure 2. In this section, we describe the development of FDR3. This summarizes work contained in several academic papers [8, 9].

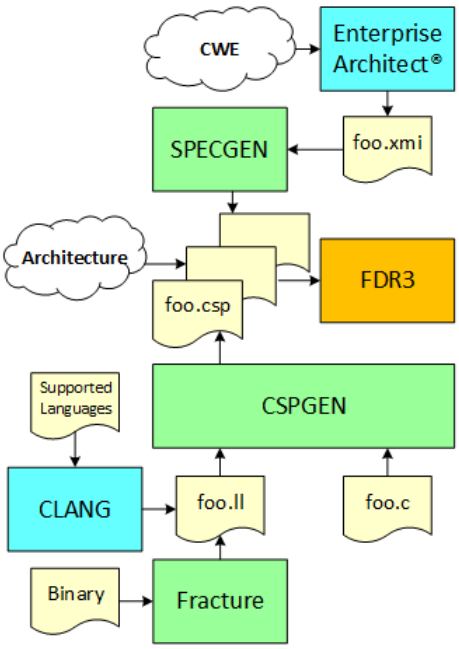

Figure 2: FDR's place in Draper's static analysis toolchain

FDR has been in continuous development since the early 1990s. Version 2 of FDR was released in 1996, and has been used broadly in academia and industry for verifying systems [10, 11]. FDR3 is a complete rewrite of FDR, funded partially by HACMS. This rewrite substantially enhanced FDR's user-friendliness and scalability as described next.

4.2.1 Language and user-friendliness improvements

One major enhancement in FDR3 is the design of a new, *statically-typed* version of machine-readable CSP. Previous versions of FDR used an untyped CSP input language, which allowed many CSP scripts with subtle errors that could only be detected during model checking and were hard to trace back to their source. The new type checker permits the vast majority of reasonable CSP programs, while ruling out many incorrect programs and keeping errors readable.

The type system resembles that of a simply-typed functional programming language, extended with base types and constructors for CSP primitives. For example, the base types include **`Event`**, the type of elements of the current program's alphabet, and **`Proc`**, the type of processes. Type constructors like "**`a => Event`**" describe events that are parameterized by data of type **`a`**. The language also includes type constructors for several standard classes of datatypes, like lists, maps,

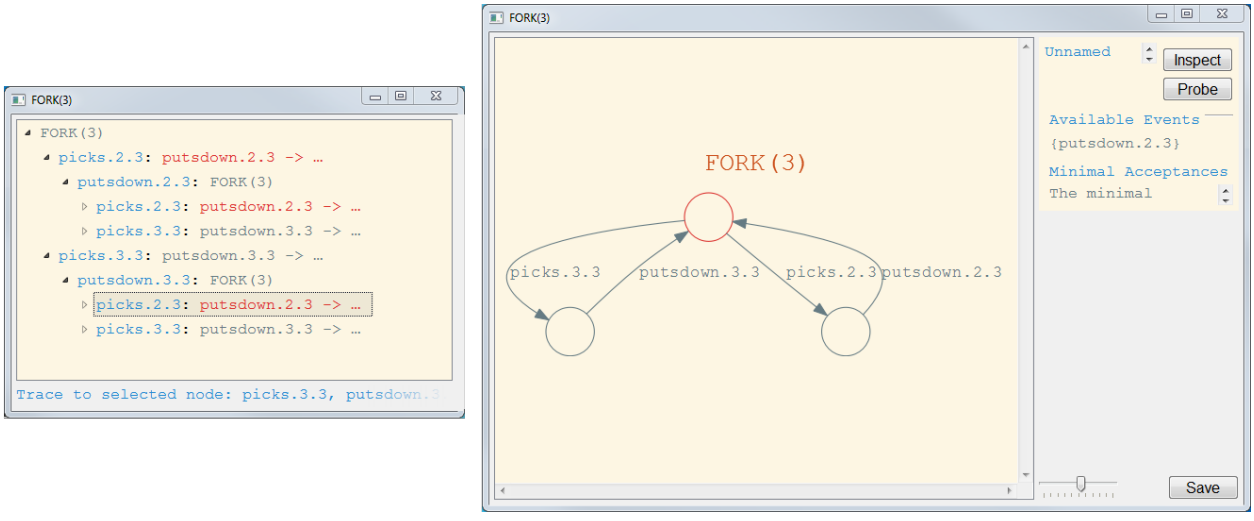

Figure 3: Example of FDR3's interfaces for interactively exploring processes

and sets.

FDR3 also includes a new graphical user interface, with support for interactively animating and exploring the behavior of processes. Figure 3 shows two examples of graphical views, using a "fork" process from the Dining Philosophers problem as an example. On the left is the "probe" interface, which allows the user to explore what events are available after a trace. On the right is the "graph" interface, which creates a graphical representation of a process. Users can select nodes in the graph to see information about possible system states.

4.2.2 Scalability improvements

FDR3 includes a completely new backend, designed to support refinement checks on processes with orders of magnitude more states than FDR2. This is achieved in two ways: First, the core data structures used in the refinement checking have been heavily optimized. Second, the core refinement checking algorithm has been *parallelized* and *distributed.* This allows it to take advantage of modern multi-core processors and of cloud computation platforms like Amazon Elastic Compute Cloud (EC2). This section describes the core refinement checking algorithm and how it was parallelized in FDR3. Section 4.2.3 describes how the algorithm was enhanced to allow distributing a check across a cluster of computers, and Section 4.2.4 provides data showing the orders-of-magnitude improvements compared with FDR2.

Refinement checking in FDR3 occurs in two steps.

Step 1: Compilation.

The first step is to compile the implementation and specification processes to *generalized labelled transition systems* (GLTSs). A GLTS is similar to a standard labelled transition system, but also allows nodes to be labelled with information related to the particular semantic model of CSP in use.

FDR3 supports two internal GLTS representations, also called *machines*, with various tradeoffs. The *Explicit* machine is a standard graph data structure, where nodes are states in the represented process and are stored in a sorted list. The *Super-Combinator* machine represents a GLTS by a series of component GLTSs along with a list of rules to combine the transitions of the components. Here, process states are represented as tuples, with one entry for each component machine.

The advantage to super-combinator machines is that the GLTS is not explicitly constructed. For example, a super-combinator representation for two parallel processes can be constructed almost instantly from the representations of the components, while an explicit representation could require considerable time to construct since the Cartesian product of the processes would need to be formed. The disadvantage of super-combinator machines is that it is slower to explore the transitions.

FDR3 supports four *strategies* for compiling processes. The strategies differ in the processes they support and the GLTS representation generated. The simplest strategies are the *low-level* and *high-level* strategies, which directly interpret the operational semantics of CSP to produce explicit or super-combinator representations, respectively. However, the high-level strategy does not support recursive processes. To mitigate this, the *mixed-level* hybrid strategy uses the high-level strategy on non-recursive components and the low-level strategy on recursive components, then wraps

them in a super-combinator. Finally, the *recursive high-level* strategy, which is new in FDR3, compiles to a super-combinator machine and supports many well-behaved classes of recursion.

Different CSP operators are most efficiently represented by different GLTS machines. When selecting a compilation strategy, FDR attempts to compile each operator at its preferred level, falling back to the low-level strategy when the environment of a process does not permit its preferred level. We have found that the recursive high-level strategy, which was unavailable in FDR2, has dramatically decreased compilation time on many examples.

Step 2: Exploration.

Once the implementation and specification have been compiled, refinement checking between the two GTLS machines begins in earnest. This check consists of an exhaustive search over the implementation GLTS, confirming that every implementation state is compatible with every specification state reachable by the same sequence of events. This search is done in breadth-first order, which produces minimal counterexamples when the check fails.

FDR2's implementation of this search was single-threaded. The algorithm keeps track of three sets of states: *current*, *next*, and *done*. These sets are represented as B-Trees, which allows the checks to efficiently use disk-based storage when RAM is exhausted. This brings the additional benefit that inserts into *done* (from *current*) can be performed in sorted order. Since B-Trees perform almost optimally under such workloads, this makes insertions into *done* highly efficient. To improve efficiency, inserts into *next* are buffered, with the buffer being sorted before insertion.

Parallelizing this algorithm essentially reduces to parallelizing the breadth-first search of the composed machine. To accomplish this, FDR3 partitions the state space based on a hash of pairs of implementation and specification machine nodes. Each available thread is assigned a partition, and has local *current*, *next*, and *done* sets. As before, memory usage is a primary concern, and becomes even more critical in a parallel setting. For example, with 16 cores, FDR3 can visit up to 7 billion states per hour, consuming 70GB of storage. Thus, checks will exceed the available RAM, and B-Trees are again a natural choice for storing these sets.

All access to the *done* and *current* sets of a given thread are restricted to that thread. However, one thread may need to insert node pairs into the *next* set of another thread (the one whose partition includes that pair). An obvious approach to support thread-safe access to the *next* sets would be locks, but considering the volume of data and the way hashing distributes the pairs across threads, this is likely to be extremely inefficient.

Therefore, instead of locks, we have generalized the buffering that is used to insert into *next* in the single-threaded algorithm. Each thread maintains a buffer for each other thread, and a list of buffers for its own *next* set received from other threads. When a buffer fills, it is immediately transferred to the target thread's list. Each thread periodically checks its incoming list, and when it reaches a certain size a bulk sort and insert operation is performed.

Experimental results (Section 4.2.4) indicate this algorithm can achieve a near linear speed up as the number of worker threads grows.

4.2.3 FDR in the cloud

Section 4.2.2 described improvements that allow FDR3 to make use of a multicore processor and achieve substantial improvements in scalability. This section describes additional enhancements

that allow FDR to make use of networked clusters of machines, rather than just multiple cores on a single processor. These enhancements enabled the use of Amazon's EC2 cloud computing platform to check a model with 1.2 trillion states and requiring 6 TB of total storage. The ability to distribute this computation across 64 16-core machines in the cloud made it possible to check this model in 5 hours.

The abstract algorithm employed by the cluster implementation of FDR is essentially the same as the parallel algorithm described in Section 4.2.2, but differs significantly in implementation details.

Each machine in the cluster runs a single FDR process, using the algorithm described above to distribute work to its individual cores. These threads still maintain *next* buffers for each other worker on the local machine. To reduce overhead, buffers for remote threads are reduced to one per remote machine. When one of these remote buffers fills, it is passed to a special thread called the *controller*, which sends it to the appropriate remote machine. The controller also receives *next* buffers from other machines, and must sort these into separate buffers for each local thread.

The most obvious potential issue with this technique is the amount of network bandwidth required to send and receive the *next* buffers. On a 16-core server, we have observed FDR3 visiting up to 30 million transitions per second. With each state pair costing 16 bytes to store, this would require 3.6 Gb/s second to be sent and received on each compute node. Data transferred between machines is compressed, which reduces the requirement to approximately 2 Gb/s.

Clearly, a commodity 1-gigabit connection is not sufficient to sustain such a volume of messages. However, a 10-gigabit ethernet connection (which are becoming increasingly common) is not only sufficient, but leaves more than enough for transient increases in rate and for future increases in processor speed or, more likely, the number of cores per machine.

The above suggests that the individual network connections are sufficient, and thus it remains to consider the total volume of data that is flowing through the network. This could be problematic: in a 64-node cluster, if each machine is sending (and receiving) 2 gigabits per second, this requires the network to be able to deal with a total of 28 Gigabytes per second. Thankfully, many modern data centers use full-bisection networks, which allow each compute node to send and receive at the maximum rate no matter what else is occurring on the network. One common network architecture is a fat-tree arrangement where the network is arranged in a tree, but the links increase in bandwidth going up the tree in such a way to ensure that all nodes have sufficient bandwidth.

Thus, in practice, while distributed FDR3 makes very heavy use of the network, recent developments in network design mean that FDR3 does not saturate it. As the number of cores increases per node, this may change, but network bandwidth is also equally likely to increase.

4.2.4 FDR benchmarks

In this Section, we describe experiments performed to measure the impact of the performance improvements described in the previous two sections. We use models of several systems as examples:

- `bully.n` is a version of the "Bully" algorithm from Chapter 14 of [3].
- `cuberoll.0` is a puzzle based on rolling 8 cubes around a 3 x 3 square.

- `ddb.n` is the distributed database example described in Chapter 15 of [2].
- `knightex.n.m` is a puzzle involving swapping pegs from colored regions of a *n* x *m* board.
- `phils.n` is the dining philosophers problem, with *n* philosophers.
- `solitaire.n` is a model of a solitaire peg-jumping puzzle. The version where *n* = 0 is a standard 33-peg puzzle. This was considered too large for FDR at the time [2] was published, but is now too small for our distributed cluster experiments. The *n* = 1 and *n* = 2 versions add 1 or 2 additional rows at the end of the four edges of the puzzle.
- `tnonblock.n` is a timed version of the non-blocking ring system from Chapter 4 of [3].
- `bakery.n.m` is a CSP file generated from an implementation of a mutual exclusion algorithm due to Lamport, and found in Chapter 18 of [3]. This is the largest example we considered – `bakery.6.30` is the trillion state example described above.
- `knightstour.n.m` is a straightforward coding of a system that explores all possible knights' tours on an *n* x *m* board.

Table 1: Comparing FDR2, FDR3 with 1 thread, and FDR3 with 32 threads.

Input	States (10^6)	Transitions (10^6)	Time (s) & Memory (GB)		
			FDR2	FDR3-1	FDR3-32
bully.7	163	1701	2205 (4.8)	1558 (3.0)	147 (7.2)
cuberoll.0	7524	20065	—	—	3991 (101.7)
ddb.0	65	377	722 (1.4)	479 (0.7)	44 (2.7)
knightex.5.5	67	259	550 (1.4)	343 (0.7)	35 (2.8)
phils.9	47	387	686 (1.0)	387 (0.4)	32 (2.0)
solitare.0	187	1487	2059 (4.4)	1540 (1.8)	101 (4.8)
solitare.1	1564	13971	19318 (35.1)	13730 (13.7)	986 (23.2)
solitaire.2	11622	113767	*	*	11003 (140.1)
tnonblock.7	322	635	2773 (6.7)	1175 (2.9)	122 (7.6)

The experiments described in Table 1 and Table 2 were performed on a Linux server with two 8 core 2GHz Xeon Chips with hyperthreading (i.e. 32 virtual cores) and 128GB RAM. Checks that took over 6 hours are marked with "–", while checks that were not attempted are marked with "*".

Table 1 compares the performance of FDR2 and FDR3 on several models, and compares the performance of FDR3 with 1 and 32 threads. There are several interesting observations. First, FDR3 with 1 worker is faster than FDR2. We believe this is because FDR3's B-Tree has been very heavily optimized, and that it makes far fewer allocations during refinement checks. FDR3 with 1 worker also uses less memory than FDR2: this is due to a new compaction algorithm used to compress B-Tree nodes that efficiently compacts sorted data by only storing the difference between keys. The extra memory used for the parallel version is for the extra buffers that are required for inserts into other workers' trees.

The speed-up that Table 1 exhibits between 1 worker and 32 workers varies according to the problem. `solitaire` is sped up by a factor of 15 (which is almost optimal given the 16 cores),

while `knightex.5.5` is only sped up by a factor of 9. The reason for this difference is the size of the iterations during the check: the time spent waiting for other workers at the end of the iteration is a larger percentage of the overall time when there are many iterations (as in `knightex.5.5`).

Table 2: The scaling performance of FDR3.

Input	Time (s) FDR3 (workers)					
	1	2	4	8	16	32
`bully.7`	1558	814	441	252	172	147
`solitare.0`	1540	786	439	236	127	101
`tnonblock.7`	1175	671	362	210	124	122

Table 2 shows FDR3's per-core scaling in more detail. As summarized above, performance scales nearly linearly with the number of threads, up to 16. As the machine in question had only 16 physical cores, performance improved less when increasing to 32 threads.

Measuring the performance of FDR when distributed in the cloud, as described in Section 4.2.3, required larger examples. Table 3 shows the size of the examples we used for cluster experiments, in terms of the number of states and transitions each model contains, as well as the amount of memory consumed. Experiments on these examples were run on Amazon's Elastic Compute Cloud. This service allows the user to rend machines of varying size on-demand. On EC2, we utilized clusters of up to 64 r3.8xlarge machines, each of which had two 8-core 2.6GHz Intel Xeons and 240GB of RAM. The machines are connected using a 10-gigabit network.

Table 3: Size of examples used in cloud experiments

Input File	States (10^9)	Transitions (10^9)	Memory (GB)
`bakery.6.8`	21.3	119.1	102
`bakery.6.30`	1174.7	6585.0	6100
`bully.8`	5.9	66.9	33
`cuberoll.0`	7.5	20.1	35
`ddb.10`	51.1	352.0	406
`knightex.3.11`	19.8	67.3	154
`knightex.5.7`	81.2	355	632
`knightstour.5.9`	123.9	207.3	574
`solitaire.1`	1.6	14.0	7
`solitaire.2`	11.6	113.8	50

Table 4 and Table 5 summarize the absolute and relative time taken for refinement checks using FDR in the cloud. Table 4 shows the absolute time that each check took on each cluster, in seconds. Table 5 shows the speedup factor that a given cluster provided for a given model, relative to the next-largest cluster. In these tables, the † symbol indicates that the check required more memory than was available on the given cluster. As Table 5 shows, on EC2 FDR3 achieves an average speedup of 67 over a single server on a 64-machine cluster, which equates to a speedup of over 1000 compared to the sequential version. Surprisingly, this is a super-linear speedup. We believe that this is because the size of the B-Trees decreases as the cluster size increases, meaning that any given B-Tree block is more likely to remain in the cache between accesses.

Table 4: Absolute time taken by cloud refinement experiments

Input File	Time by Cluster Size (s)						
	1	2	4	8	16	32	64
`bakery.6.8`	10069	5919	2940	1378	675	356	186
`bakery.6.30`	†	†	†	†	†	37701	20680
`bully.8`	4183	2502	1186	532	249	113	57
`cuberoll.0`	1439	870	425	206	98	49	25
`ddb.10`	†	29389	15329	6711	2801	1337	562
`knightex.3.11`	7259	4622	2257	1059	489	229	112
`knightex.5.7`	†	†	15987	7195	2954	1453	586
`knightstour.5.9`	†	†	12165	5926	2201	1091	441
`solitaire.1`	568	356	174	99	52	21	12
`solitaire.2`	5383	3726	1982	920	439	190	90

Table 5: Speedup factor scaling in cloud refinement experiments

Input File	Speedup Factor by Cluster Size						
	1	2	4	8	16	32	64
bakery.6.8	—	1.70	2.01	2.13	2.04	1.09	1.91
bully.8	—	1.67	2.11	2.23	2.14	2.19	1.99
cuberoll.0	—	1.65	2.05	2.06	2.10	2.02	1.92
ddb.10	†	—	1.92	2.28	2.40	2.10	2.38
knightex.3.11	—	1.57	2.05	2.13	2.17	2.14	2.04
knightex.5.7	†	†	—	2.22	2.44	2.03	2.48
knightstour.5.9	†	†	—	2.05	2.69	2.02	2.48
solitaire.1	—	1.59	2.05	1.76	1.89	2.48	1.80
solitaire.2	—	1.44	1.88	2.16	2.09	2.31	2.11
Average	—	1.61	2.01	2.11	2.22	2.13	2.09
Average vs 1 Node	—	1.61	3.23	6.82	15.12	32.23	67.43

Table 5 also indicates that the cluster version imposes a small overhead, since the average speedup from one to two nodes is 1.61. Some of this slowdown will be because the state pair blocks must be compressed before being sent to remote nodes, but the source of the remainder is unclear to us. Thanks to the superlinear scaling observed above, this effect is cancelled out with clusters of 32 compute nodes or more.

4.3 SpecGen Research and Development

SpecGen is a Draper-developed tool that translates statecharts drawn in the commercial Enterprise Architect modeling tool to CSP. These CSP translations may be analyzed directly to verify properties of a system model, or may be used as a specification to compare with a CSP model extracted from a source code implementation. Figure 4 shows SpecGen's place in our static analysis toolchain. This section describes the research Draper performed in the development of SpecGen, and provides an example of its use, expanding on an academic research paper published during the HACMS program [12].

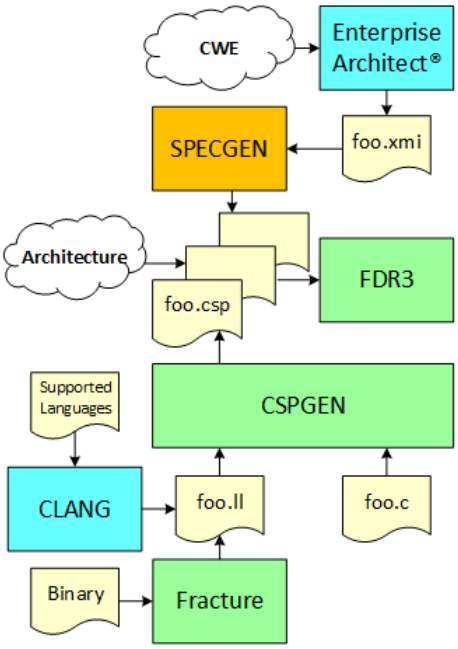

Figure 4: SpecGen's place in Draper's static analysis toolchain

Statecharts are a widely-used technique for graphically representing the high-level behavior of complex systems. Since their introduction by Harel [13], support for various versions of statecharts has been implemented in many commercial tools, including Enterprise Architect and Simulink Stateflow. As the use of statecharts has become widespread, so too have techniques for formally verifying their behavior. Classic examples include modeling via translation to SPIN [14] or Symbolic Model Verification [15].

Translating statecharts to CSP has two main advantages. First, as discussed above, CSP is a rich, expressive language for writing specifications. We may leverage FDR to check these specifications and to interactively explore the behavior of the translated systems. Second, the other

tools in Draper's static analysis toolchain already use CSP as a common modeling language. Statecharts are a convenient way to represent specifications for more complex systems already translated to CSP (e.g., by CspGen). In this context, statecharts provide an intuitive, graphical common language for specifications. This common language can be used to agree on specifications with a domain expert who implemented a system in C, but is not familiar enough with CSP to write formal specifications directly.

The SpecGen tool builds on previous work for modeling statecharts in CSP [16]. During HACMS, we have added support for several additional statechart features and designed a new, simplified algorithm by using new CSP language constructs, described in Section 4.3.2. The tool supports statecharts developed with Enterprise Architect and is the first practical implementation of any such translation. The SpecGen distribution also includes several examples, described in Section 4.3.1, and is available freely under a permissive open-source license [17].

4.3.1 A Statechart Example: The Dining Philosophers

To illustrate the use of SpecGen, we consider the classic dining philosophers problem [1]. Our distribution of SpecGen includes this example, implemented as a statechart in Enterprise Architect, for 2, 3 and 4 philosophers. Figure 5 shows statecharts representing Philosopher 2 and Fork 2 from the four-philosopher system. We elide the full system for clarity – it consists of four philosophers and forks, like those shown, as parallel sub states of one top-level node.

We begin our explanation with the statechart for Fork 2. Conceptually, it keeps track of which philosopher has permission to use the fork at any time. It begins in the state Free, indicating that the fork is not in use and may be claimed by either philosopher. Transitions to the **`Phil2Holds2`** and **`Phil3Holds2`** states are guarded by the constraints **`In(WaitingRight2)`** and **`In(WaitingLeft3)`** respectively. This ensures these transitions are not taken until the relevant philosopher is in the state where he is waiting on this fork, so the ownership of the fork is not given to a philosopher until he wants it.

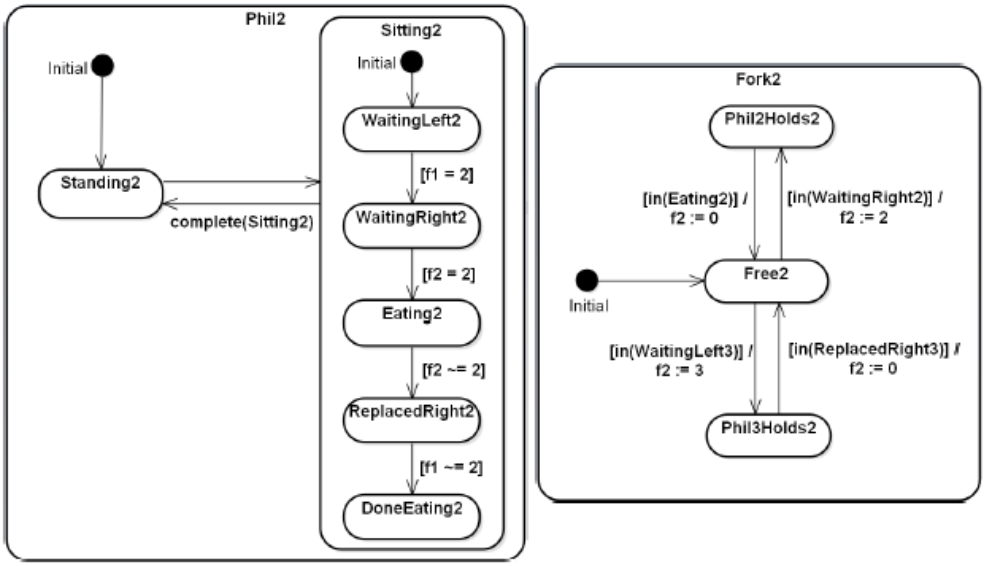

Figure 5: Statecharts for one philosopher and one fork

The system also includes four variables, **f1, ..., f4**, one for each fork. Intuitively, the value in these variables indicates which philosopher, if any, currently has permission to use a given fork. Thus, the transition from state **Free2** to state **Phil3Holds2** sets variable **f2** to **3**. These variables are set by the forks, and used by guards in the philosophers. For example, consider node **WaitingLeft2** in **Phil2**. This node models the state where Philosopher 2 is waiting to pick up his left fork (Fork 1). The guard on this transition prevents it from being taken unless **f1 = 2**, indicating that Philosopher 2 has permission to use Fork 1. Similarly, the transition from **Eating2** to **ReplacedRight2** is guarded by the requirement that **f2** is not **2**, indicating that Philosopher 2 no longer has permission to use his right fork. The semantics of statecharts require that all available transitions are taken immediately, ensuring that Fork 2 and Philosopher 2 remain synchronized here.

Finally, we consider the edge from **Sitting2** back to **Standing2**, which is labeled with the completion event **complete(Sitting2)**. In statecharts, events are named triggers that are often used to represent external events. During execution, a set of enabled events is provided as input, and an edge labeled with an event may only be taken if the event is currently enabled. Completion events are special events that are enabled when a node terminates, rather than by input. A node is considered to have terminated when all its concurrent subnodes have reached states with no out-edges. Here, the event label prevents the philosopher from standing until he is done eating.

It is worth noting that this example is not intended to represent the most efficient or natural implementation of the dining philosophers as a statechart. Rather, we have designed it to highlight several features supported by the tool.

The Generated Model

When run on an Enterprise Architect statechart like the one described above, SpecGen produces several files containing CSP definitions, including a top-level process **RunSystem** that models the statechart's behavior. The behavior of a CSP process is most easily described by finite "traces" of observable events. In the case of **RunSystem**, the relevant observable events include:

- **transition.N.E**, indicating a transition between nodes. Here **N** is the name of the node that contains the transition, and **E** is the name of the edge itself. Typically, SpecGen will generate node names that match the name given in the statechart if all nodes have unique names, and will otherwise pick a name based on the full path of a node. Edges are given names like **Node1__Node2**, indicating a transition from **Node1** to **Node2**.
- **tock**, indicating the completion of a "step" of the statechart. According to the semantics of statecharts, a step comprises a single transition in every currently-running subchart that can make one.
- **read.x.n** and **write.x.n**, indicating reads or writes of a value **n** in variable **x**.
- **writeerror.x**, indicating that the statechart has a race condition where two parallel subcharts attempted to write to the variable **x** in the same step.

Finding the Deadlock

The most obvious property to check in the dining philosophers example is deadlock freedom. In our CSP scripts, this property is stated:

```
assert RunSystem \ {| tock |} :[ deadlock free ]
```

The **** ("hiding") operator here is used to hide the tock events of **RunSystem**. A statechart continues to take "steps", represented by these events, even if no subchart can make a transition. Intuitively, to detect the deadlock, we must inform FDR that the mere passage of time does not count as progress.

Asking FDR to check this property results in an assertion failure, as expected. Indeed, because the semantics of statecharts require each parallel process to make a transition in each step if able to, this system will always deadlock. FDR also displays the trace that leads to the deadlock. For the three-philosopher system, this trace ends with the events:

```
transition.Sitting2.WaitingLeft2__WaitingRight2 ,
transition.Sitting3.WaitingLeft3__WaitingRight3 ,
transition.Sitting1.WaitingLeft1__WaitingRight1
```

We see that the last three events are each philosopher transitioning to his **WaitingRight** node, indicating that each philosopher has picked up his left fork and is waiting on his right fork.

More Complicated Properties

While checking for deadlock is useful, the real power of FDR comes from its ability to write more interesting specifications as processes and check that these hold via refinement. As an example, we consider the following property: "after sitting, no philosopher stands without eating". In this section, we will demonstrate how to state and check this property for the 3-philosopher system, and show how an error in the statechart could be caught.

A convenient way to check that a trace never occurs in a system is to use a "watchdog process" [3]. The idea is to build a process that recognizes the disallowed sequence and issues an error event if it occurs. This "watchdog" may then be synchronized with the system under test, and a refinement check may be used to see if the composed system ever issues the error event.

We begin by identifying the events of interest for our property. We define functions **sitEvent**, **eatEvent**, and **standEvent**, which identify the transitions on which a philosopher sits, stands, or eats, respectively. We show only **sitEvent**:

```
sitEvent :: (Int) -> Event
sitEvent(1) = transition.Phil1.Standing1__Sitting1
sitEvent(2) = transition.Phil2.Standing2__Sitting2
sitEvent(3) = transition.Phil3.Standing3__Sitting3
```

Next, we define the error event that will be thrown if a philosopher stands without eating. It is

parameterized by the number of the philosopher so that we may see who transgressed:

```
phils :: {Int}
phils = {1,2,3}


channel stoodTooSoon : phils
```

We implement a philosopher's watchdog as a pair of mutually recursive processes. The first process, **watchStanding**, waits for a philosopher's "sit" event and transitions to **watchSitting**. The **watchSitting** process waits to see whether an "eat" event or a "stand" event comes next. If "eat" occurs first, it waits for the "stand" event and then returns to **watchStanding**. If "stand" occurs first, it throws the error. The top-level watchdog is then the parallel composition of the watchdogs for each philosopher:

```
watchStanding, watchSitting :: (Int) -> Proc
watchStanding(i) = sitEvent(i) -> watchSitting(i)


watchSitting(i) =
    (eatEvent(i) -> standEvent(i) -> watchStanding(i))
 [] (standEvent(i) -> stoodTooSoon.i -> STOP)


WatchDog :: Proc
WatchDog = ||| i <- phils @ watchStanding(i)
```

We define a set **evs** of the events of interest for our property. The original system and the watchdog are placed in parallel and required to synchronize on the events in **evs**, so that the watchdog can keep track of the system as it executes.

```
evs :: {Event}
evs = { sitEvent(i), standEvent(i), eatEvent(i) | i <- phils }


WatchdogSystem :: Proc
WatchdogSystem = (RunSystem [| evs |] WatchDog)
```

Finally, we state the property that the error event can never occur in the composed system. This uses the CSP operator **|** ("projection"), which is the opposite of the hiding operator we saw above – only the projected events are visible. The assertion says that the system where only the error event is visible is a refinement of **STOP**, the system which performs no events.

```
assert STOP [T= WatchdogSystem |\ {| stoodTooSoon |}
```

FDR verifies that this property holds. However, suppose we had made a mistake and left off the guard on the edge from **`Sitting2`** to **`Standing2`**. According to the StateMate semantics of statecharts, transitions between higher-level nodes are preferred when a choice is available. So, the modified chart will transition out of **`Sitting2`** immediately after entering it. When we ask FDR to check the property for this modified version of the chart, it reports:

```
Result: Failed

Error Event: stoodTooSoon.2
```

Performance

Table 6: Time to find deadlock in CSP models generated from statecharts

Philosophers	2	3	4
Time	2s	6s	117s

The time to find the deadlock in FDR is summarized in Table 6, organized by the number of philosophers in the system. These times are the averages of 5 runs performed on an Intel Xeon E5-2630 v3. The machine had 32GB of RAM, but all tests consumed less than 6 GB.

Predictably, the time to find the deadlock grows exponentially with the number of philosophers. Checking these translated statecharts is slower than checking more natural implementations of the dining philosophers in CSP, because accurately modeling the semantics of statecharts involves substantial coordination overhead and additional features like per-node timers. As statecharts offer the advantage of wider accessibility, we believe this overhead is sometimes justified.

There was not time within the HACMS program to investigate substantial performance improvements in the SpecGen output models. The current version of SpecGen generates models that were designed with the primary goal of semantic fidelity, not speed of model checking. For these reasons, we believe it will be possible to improve the efficiency of these models in the future.

4.3.2 Translation Enhancements

As mentioned above, SpecGen builds on an earlier algorithm for modeling statecharts in CSP, by Roscoe and Wu [16]. In addition to providing a practical implementation, we have improved on that paper's translation by including support for two additional statechart features (the “in” guards and completion events described in Section 4.3.1) and exploiting a newer FDR feature to simplify the generated models. The remainder of this section describes this simplification.

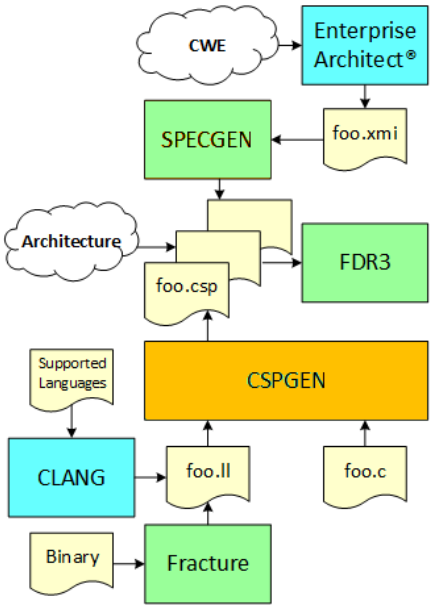

Figure 6: CspGen's place in Draper's static analysis toolchain

The biggest challenge in modeling statecharts in CSP is representing *priority*. In CSP, a process may select freely among its available actions, but in statecharts certain transitions may be favored over others. For example, nodes must be allowed to take an “idle” step if and only if no transitions are available. Also, transitions out of a state may be favored over transitions within that state when both are available, or vice versa – classic Statemate semantics [18] favor outer transitions while Unified Modeling Language (UML) favors inner ones [19]. (In SpecGen we have followed

[16] in modeling Statemate, but it would be straightforward to prefer the alternate order, which is more common today).

Roscoe and Wu's translation models these instances of priority with a subtle renaming and synchronization scheme [3]. Happily, modern versions of FDR include a new feature that SpecGen uses to simplify this: **`prioritise`**. This function takes as arguments a process **`P`** and an ordered list **`evs`** of sets of events. If **`P`** may perform events from different sets in **`evs`**, then **`prioritise(P,evs)`** may perform only events from the first set that contains any of **`P`**'s events. Combining **`prioritise`** with interrupts, where a CSP process may be preempted by certain events, also allowed for a simplified encoding of "promoted" actions in statecharts. These actions allow an inner node to transition directly to an outer node, terminating its parallel siblings.

4.4 CspGen Research and Development

CspGen is a Draper tool that builds CSP models of imperative programs. Figure 6 shows its place in our static analysis toolchain. These models may be analyzed using the FDR3 model checker to check properties of the initial program. Draper uses this tool in the analysis of software provided by the other performers, and has made it available as free, open-source software [20]. This distribution comes with many example programs and specifications, which can be used to explore the concepts described in this section in more detail.

The initial version of CspGen supported C source code as input. The tool now also accepts the Low-Level Virtual Intermediate Representation. Since many programming languages can be compiled to LLVM IR, this addition enables the application of the Draper toolchain to a much wider range of software.

The architecture of the tool is shown in Figure 7, using C source code input as an example. CspGen parses the C file and then generates two CSP source files from it: a memory model and a functional model. The memory model captures information about the state that is used in the execution of the C program, like global variables and stack variables. The functional model captures the operational behavior of the program, with reference to the memory model where appropriate. Finally, these are combined with a "runtime" or "environment" model. This last piece captures information about the environment in which the C program expects to run, like libraries it uses and available hardware. These three models are composed to form a complete model of the C program's behavior, which can then be analyzed in FDR.

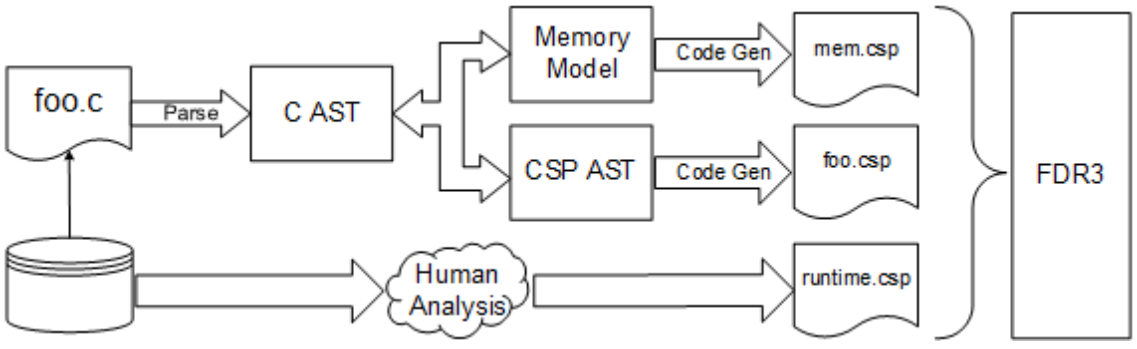

Figure 7: CspGen Architecture

In the remainder of this section, we describe in greater detail CspGen's model of operational semantics and state in imperative languages (Sections 4.4.1 and 4.4.2) and describe the formal verification of CspGen's core translation in Coq (Section 4.4.3).

4.4.1 CspGen's Model of Addressable State

The first question to answer when modeling an imperative program in CSP is how to represent state. CSP is purely functional – it has no notion of mutable variables. The traditional way to represent mutable variables is, therefore, via a process that runs in parallel with the main operational process; and can be communicated with via *read* and *write* events [2]. For example, the process **`VAR(x)`** below represents a variable with current value **`x`**:

```
VAR(x) = (read!x -> VAR(x)) [] (write?y -> VAR(y))
```

This process can be communicated with via the event **`read.x`**, where **`x`** is the current value of the variable, or via the event **`write.y`**, which changes the current value to **`y`**. This representation could be scaled to programs with *n* variables by putting *n* such variable processes in parallel with the main operational process, and give each variable process its own *read* and *write* event channel names.

One problem with such a representation is that it does not offer a natural way to take the address of a variable. Since we are working with imperative languages like C, which include *pointers*, we need to support this operation.

It might seem natural to use the distinct *read* and *write* event names for each variable as an address representation. However, channel names are not first-class data in CSP. Instead, we use only one global *read* channel and one global *write* channel, but add an extra parameter that indicates which variable is being read from or written to. This extra parameter can be thought of as the address of the variable:

```
VAR(addr,x) =    (read!addr!x -> VAR(addr,x))
              [] (write!addr?y -> VAR(addr,y))
```

Then memory is the parallel interleaving of all addresses. CspGen calculates the number of addresses needed and builds, roughly, this process:

```
ALL_VARS = ||| {(addr,init_val) <- ALL_ADDRS} @ VAR(addr,init_val)
```

Dynamic allocation can be supported in this model by creating extra addresses at translation time, and including an "allocator" processes that accepts *alloc* events from the main operational processes and returns address ranges. However, the software we encountered in HACMS used only statically allocated memory, so this was not necessary.

A remaining issue with this representation is that it is relatively inefficient. Each variable process will have a state for each possible value of that variable. While FDR has been demonstrated to support processes with trillions of states (as described in Section 4.2.4), it does not take many 32-bit variables to reach this limit. This problem can be mitigated in several ways:

- A traditional static analysis technique, like abstract interpretation, can be used to bound the possible range of variables before translation.

- We can support only a limited number of fixed values, and add an "unknown" value that introduces nondeterminism when examined.
- We can use static analysis to determine which variable addresses can escape their scope at runtime, and use a cheaper, more local model for variables with limited scope.

In practice, CspGen uses a combination of the second and third technique. We discuss the cheaper, local model for non-escaping stack variables in Section 4.4.2.

4.4.2 CspGen's Model of Imperative Control Flow

The next question to answer is how to support the control flow of an imperative program. In this section, we write **`|c| => P`** to indicate that a statement **`c`** written in the C language is translated to the CSP process **`P`**.

Consider compound C statements like "**`c1; c2`**" where **`c1`** and **`c2`** are C statements. CSP has a native notion of termination SKIP and a sequence operator "**`;`**", so it would be natural to translate such an expression this way:

```
|c1; c2| => |c1|; |c2|
```

This representation presents at least two problems. First, no information is passed from the translation of **`c1`** to the translation of **`c2`**. This may not be a problem if all state is stored in globally accessible parallel processes like those described in the previous section. But, as discussed there, this can be quite inefficient – we'd prefer a more local way of communicating local effects (like writing to thread-local variables) between **`c1`** and **`c2`**. Second, C includes features that circumvent the normal control flow, like **`break`**, which might occur in **`c1`**. Because there is no analog to this non-local control flow in CSP, this representation provides no way to model these C features.

To solve both these problems, we borrow a standard trick from compilers: continuation-passing style [21]. Rather than translating a C statement as a simple CSP process, we translate it as a CSP *function* with two arguments. The first argument is the current local state. The second argument is a *continuation*: another function describing what should occur after the current command ends. The continuation itself expects to be passed a copy of the updated local state. Thus, the translation becomes:

```
|c1; c2| => \(st,cont) @ |c1| (st, \st' @ |c2| (st',cont))
```

Non-local control flow is now naturally supported, because the translation of **`c1`** or **`c2`** may simply ignore its continuation argument.

4.4.3 Formally Verifying the Core Algorithm of CspGen

The purpose of CspGen is to construct a faithful model of an imperative program in CSP so that it can be analyzed with FDR. If this model is inaccurate, then the results of the analysis cannot be trusted. Since the model described in the previous two sections is complicated, such inaccuracies could easily be missed. To prevent these problems, the Draper team performed a formal verification of the soundness of the core translation from imperative programs to CSP.

This verification was performed in the Coq interactive theorem prover [22]. Proving soundness of a complete translation from C to CSP would be a task too large for the scope of the HACMS

program, so we instead verified the core model of imperative programs using a simpler source language. This relatively standard language, which we call **`While`**, is based on the Imp chapter of the Software Foundations textbook [23] (which is itself based on Winskel's classic introduction to the semantics of imperative langauges [24]). We describe the formal proof in this section, assuming a basic knowledge of Coq. It is also included with our CspGen distribution.

The first step in this task was to create a formal definition of each language, including a semantics describing the meaning or behavior of programs in the language. In both cases, we defined a relatively standard "small-step" operational semantics. This is a relation describing how expressions from the language are transformed by small steps of computation, which can then be strung together to completely execute a program. For CSP expressions, this relation had four arguments:

```
Step : Env -> Proc -> Event -> Proc -> Prop
```

The first argument, of type **`Env`**, is an "environment" that assigns a CSP process to each variable. The second argument, of type **`Proc`**, is the original CSP process. The third argument, of type **`Event`**, is the CSP event that occurs in this step of computation (or the special event "**`tau`**" if no observable event occurs). The final argument, of type **`Proc`**, is the transformed process after a step of computation.

Based on **`Step`**, we can define the "traces" of a CSP processes as the lists of events that can occur by a series of steps from a given process:

```
OpSemTraces : Env -> Proc -> Trace -> Prop
```

For commands in the While language, we defined a similar step relation:

```
CStep : cmd -> state -> option wevent -> cmd -> state -> Prop
```

This relation differs in a few ways. **`While`** commands have the type "**`cmd`**" in our Coq formalization. Unlike CSP, where there is one input environment, this relation has two "**`state`**" arguments because execution of the command may change the state. Finally, the **`While`** event argument (of type **`wevent)`** is optional, because the **`While`** language does not have a natural notion of "uninteresting" event like **`tau`** for the case where the step of computation has no observable effect. Building on this, we define a "multi-step" relation that formalizes the idea of a series of steps:

```
MCStep : cmd -> state -> list wevent -> cmd -> state -> Prop
```

With a definition for each language in hand, we can define the translation from CSP "**`Proc`**"s to **`While`** "**`cmd`**"s, which is the core of CspGen's algorithm.

As described at the beginning of Section 4.4, this translation actually produces two distinct CSP processes from an input imperative program: a memory model and an operational model. The memory model is built by the function **`MemProc`**:

```
MemProc : nat -> state -> Proc
```

This function takes as arguments a natural number, indicating the number of variables used by the **`While`** program, and the program's initial state. It produces a CSP process representing the

memory used by the program, and intended to be put in parallel with operational process, synchronizing on memory reads and writes.

The operational model is built by the function **compile**:

```
compile : cmd -> Proc -> Proc
```

Since we use a continuation-based translation, as described in Section 4.4.2, this function takes not only the command to be translated, but also a CSP process representing its continuation.

These functions generate the memory model and operational model of a command, respectively. The results are intended to be put in parallel, synchronized on memory events. In this simplified model, memory events are the only events, so it is enough to synchronize on all events. So the complete translation of a **While** program **wprog** with initial state **st** in this Coq implementation is:

```
PGenPar (compile wprog PStop)
        allEvents
        (MemProc (fvs_cmd wprog) st))
: Proc
```

Here, **fvs_cmd** counts the number of variables used in a **While** command, and **PGenPar** is the Coq formalization of CSP's "generalized parallel" operator. Its first and third arguments are processes that are put in parallel, synchronized on the set of events given as the second argument.

With the Coq formalization of the translation in hand, it is time to define and prove soundness of the translation. The intuitive soundness property we'd like to capture is that any trace of "**wevent**"s that can occur in a valid execution of a **While** program is mirrored by a similar trace of "**Event**"s in the CSP semantics of its translation. To state this property, we need to define "similar trace". Since the two event definitions are essentially just different names for reads and writes to memory, it is possible to define a direct translation:

```
whileToCSPTrace : list wevent -> Trace
```

Therefore, we might guess the appropriate correctness property is:

```
Theorem translation_sound : forall wprog wprog' st st' wtrace,
    MCStep wprog st wtrace wprog' st'
  -> OpSemTraces WhileEnv
                 (PGenPar (compile wprog PStop)
                          allEvents
                          (MemProc (fvs_cmd wprog) st))
                 (whileToCSPTrace wtrace).
```

However, this proposition is not quite true. The problem is the result of one of the state-explosion

mitigations described in Section 4.4.1. The While program execution remembers precise values for each variable, but the CSP program remembers only values within a small range. When a read or a write to a variable goes outside this range, it is replaced by a unique "unknown" value that induces non-determinism in the model. Thus, we must allow for CSP traces that are less precise in that exact values can be replaced by this "unknown" value. We introduce an approximation relation on CSP traces that captures this loss of information:

```
ApproxTrace : Trace -> Trace -> Prop
```

We can now state the correct theorem:

```
Theorem translation_sound : forall wprog wprog' st st' wtrace,
    MCStep wprog st wtrace wprog’ st’
  -> exists ctrace,
       ApproxTrace (whileToCSPTrace wtrace) ctrace
     /\ OpSemTraces WhileEnv
                    (PGenPar (compile wprog PStop)
                             allEvents
                             (MemProc (fvs_cmd wprog) st))
                    (whileToCSPTrace wtrace).
```

The definition and proof of this theorem required approximately 2500 lines of Coq proof script, after the definition of the two languages.

4.5 Fracture

Fracture is a decompiler that translates ARM binaries to the LLVM Intermediate Representation. Draper implemented a proof-of-concept version of Fracture as a modification and extension of the LLVM compiler suite and has made it publicly available [25].

The core idea of Fracture's decompilation strategy is to reverse LLVM's TableGen-based instruction selector. *Instruction selection* is one the final stages of compilation, where machine-specific instruction sequences replace machine-independent LLVM IR. A common approach to instruction selection is for human experts to populate a map data structure from IR sequences to efficient machine-specific implementations, and an algorithm is used to match every piece of the IR program with corresponding implementations from the map.

TableGen is LLVM's generic implementation of this map data structure. Fracture is implemented as a new TableGen map that reverses the map used in instruction selection, and some associated libraries. It ingests a basic block of target instructions and emits a directed acyclic graph (DAG) which resembles the post-legalization phase of LLVM's Selection DAG instruction selection process. It leverages the pre-existing target LLVM TableGen definitions, without modification, to provide a generic way to abstract LLVM IR efficiently from different target instruction sets.

An initial proof-of-concept version of Fracture was completed in 2014. Experimentation determined that the generated LLVM IR was still quite low-level, especially compared with LLVM IR generated via compilation from a source language with Clang. For example, the Fracture-generated IR often made use of machine-specific memory layout information. This made it challenging to model with our CspGen tool.

Additionally, experimentation with the Fracture prototype revealed two flaws in the instruction selection reversal technique. First, Fracture's algorithm does not preserve single static assignment (SSA) form. The means that the lifting of the binary to LLVM Intermediate Representation does not ensure that each variable to be defined and assigned once before it is used. Since code generation relies on the SSA form during phases like basic block control flow and alpha renaming, the simple map inversion implemented in Fracture was behaviorally unsound. Second, Fracture's approach required following every branch to determine whether that branch led to a function or a continuation in the intraprocedural control flow graph. This fails when following a branch to a concrete address that resolves in a branch to an address that cannot be computed without reasoning about the dataflow into the branch target. As a result, translation could terminate without examining all assembly in text sections.

At the same time, research conducted in the DARPA Cyber Grand Challenge program was beginning to result in the release and maturity of other open-source decompilation and binary analysis frameworks, like BAP [26] and Angr [27]. Thus, the decision was taken to stop work on raising the abstraction level of Fracture's output, with the intention to integrate with existing open-source decompilers in the future. As the assessments of Galois-provided software in the remainder of HACMS did not require formal binary analysis, this integration was unnecessary for HACMS, but would be an interesting avenue for future research.

4.6 Formal Assessments

Application of the formal tools described above to Blue Team code focused on Galois' Ivory/Tower language. This choice was made on the advice of the DARPA HACMS program manager, because Ivory/Tower compiles to C, a language already understood by our tools, and because these programs have an understandable and formalizable specification. We attacked the problem of verifying Ivory/Tower code from two directions: verifying as-implemented Ivory/Tower programs from their C versions, and verifying the Tower concurrency model directly. Section 4.6.1 describes an example of the first direction in more detail. This information also appears in our Formal Vulnerability Assessment Report. Section 4.6.2 summarizes other efforts.

4.6.1 Verifying an Ivory/Tower program in detail

In one assessment, we formally verified that a Galois-supplied Ivory/Tower program had the expected behavior. The program comprised two threads that caused an LED to blink. We used Draper's CspGen tool and Oxford's FDR model checker for this task. The high-level process of verifying the system followed the outline described in Section 3.3.2

Step 1: Environment modeling

Software runs in an environment of libraries, system calls, and hardware that provide services and a means to interact with the external world. To model the behavior of the software, we need a model of this ecosystem.

In the case of the Galois LED program, we built models for two main components: the FreeRTOS threading primitive used to spawn the two threads, and the hardware pins that the software manipulated to adjust the LED. The threading primitive was quite straightforward to model, considering CSP's natural support for concurrency and our previous experience modeling similar primitives from the standard Linux pthreads library. For the hardware pins, a custom model was needed.

Examination of the source code revealed that the LED is manipulated by turning on and off the current to two Universal Asynchronous Receiver/Transmitter (UART) pins (pins 14 and 15 on the relevant platform). To keep the model simple, we modeled only these two pins:

```
datatype PinState = CurrentOn | CurrentOff

datatype PinName = Pin14 | Pin15
```

We built CSP channels representing software-triggered interactions with the pins:

```
channel pin_set   : PinName

channel pin_clear : PinName

channel pin_read  : PinName.PinState
```

We built processes that stored the current state of each pin, and a wrapper to execute a program that can manipulate the pins:

```
pin :: (PinName,PinState) -> Proc

pin (nm,state) = pin_set.nm        -> pin(nm,CurrentOn)

              [] pin_clear.nm      -> pin(nm,CurrentOff)

              [] pin_read.nm!state -> pin(nm,state)


hardware :: Proc

hardware = pin(Pin14,CurrentOff) ||| pin(Pin15,CurrentOff)


runOnHardware :: (Proc) -> Proc

runOnHardware (p) =

    p [| {| pin_set, pin_clear, pin_read |} |] hardware
```

Finally, we considered the way these pins are manipulated in C code. The board on which this test was intended to run supported memory-mapped control of the pins. The generated C code simply read from or wrote to fixed addresses in memory for this purpose. To simplify modeling, we replaced these reads and writes with calls to new functions, and implemented a model for them:

```
gpiob_pin15_current_on (stubState,stubCont) =
```

```
  pin_set!Pin15 -> stubCont(stubState, UnitVal)

gpiob_pin15_current_off (stubState,stubCont) =
  pin_clear!Pin15 -> stubCont(stubState, UnitVal)

gpiob_pin14_current_on (stubState,stubCont) =
  pin_set!Pin14 -> stubCont(stubState, UnitVal)

gpiob_pin14_current_off (stubState,stubCont) =
  pin_clear!Pin14 -> stubCont(stubState, UnitVal)
```

The arguments to these functions are artifacts of the continuation-passing style used to represent C control flow in our model, as described in Section 4.4.2.

Step 2: Building a CSP specification for the program.

For this assessment, we were interested in confirming that the Galois program's behavior was to toggle the LEDs on and off indefinitely by manipulating the pins. Thus, we built a simple model of the two threads and their interleavings.

```
loop_250_spec :: Proc
loop_250_spec = pin_set.Pin14 -> pin_clear.Pin14 -> loop_250_spec

loop_333_spec :: Proc
loop_333_spec = pin_set.Pin15 -> pin_clear.Pin15 -> loop_333_spec

tower_entry_spec :: Proc
tower_entry_spec =
    pin_clear.Pin15 -> pin_clear.Pin14
                    -> (loop_250_spec ||| loop_333_spec)
```

Extracting a CSP model of the Ivory/Tower program

To extract a model of the as-implemented system, we first compiled the Galois code from Ivory/Tower to C, and then used Draper's CspGen tool to translate the C code to CSP. This resulted in a large CSP file that described the behavior of the Galois program, with calls to the environment model described above. In particular, the translated CSP version of the "**`tower_entry`**" function captures all the behavior of the system.

Verifying that the implementation refines the specification.

Finally, we wrote a CSP assertion capturing the property that the implementation is a refinement of the above specification, when restricting the set of observable events to pin manipulations:

```
assert hideMemory(runInMemory(tower_entry((| |), \_,_@STOP)))
    [T= tower_entry_spec
```

Using FDR, we checked this assertion. This confirms that the only possible behaviors of the as-implemented system are captured by the expected formal specification. Thus, no flaws were found in this Galois LED example and a formal proof has been obtained that the system behaves correctly up to the assumptions embodied by our model of computation.

4.6.2 Other Ivory/Tower verification

We applied our static analysis tools to several similar analyses. Human analysis in preparation for formal modeling identified a replay attack against an early version of some Ivory/Tower Secure Mathematically-Assured Composition of Control Modules Pilot (SMACCMPilot) communication components. We subsequently demonstrated that our tools can find this attack, both by analyzing a hand-built abstract model in FDR (as described in [28]), and by analyzing a model extracted from the generated C code with CspGen.

We also directly considered the Ivory/Tower model of concurrency. This language in intended to prevent common concurrency errors, like deadlock and race conditions, by construction. Based on a formal semantics provided by Galois, the Draper team built a model of Ivory/Tower program execution in CSP. The model is parameterized by several characteristics of a given program, like the numbers of threads and variables. By analyzing the model in FDR, we confirmed that the abstract Ivory/Tower semantics prevent the concurrency errors described above, using parameters chosen from some example programs. There are several avenues for future work in this space, like building a tool to extract the relevant parameters from an Ivory/Tower program automatically, or designing a more general model that could confirm these properties for all Ivory/Tower programs.

4.7 Academic Publications

The Draper team's work on these formal verification tools contributed to several academic publications, listed here:

Brandon Shapiro and Chris Casinghino. "SpecGen: A Tool for Modeling Statecharts in CSP," NASA *Formal Methods*, 2017.

Pedro Antonio, Thomas Gibson-Robinson, and A.W. Roscoe. "Tighter Reachability Criteria for Deadlock-Freedom Analysis," *21st International Symposium on Formal Methods*, 2016.

Pedro Antonino, Thomas Gibson-Robinson, and A. W. Roscoe. "Efficient Deadlock-Freedom Checking using Local Analysis and SAT Solving," *Proceedings of the 12th International Conference on integrated Formal Methods*, 2016.

Colin O'Halloran, "Verifying Critical Cyber-Physical Systems After Deployment," *Proceedings of the 15th International Workshop on Automated Verification of Critical Systems*, 2015.

Thomas Gibson−Robinson and A.W. Roscoe. "FDR into The Cloud," *Communicating Process Architectures,* 2014.

Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, A.W. Roscoe. "FDR3 — A Modern Refinement Checker for CSP," *Tools and Algorithms for the Construction and Analysis of Systems,* 2014.

5. CONCLUSION

This report has described the testing and research carried out by the HACMS Red Team. The team comprised a novel combination of state-of-the-art penetration testing with formal methods research.

Our formal methods research resulted in substantial improvements to the FDR CSP refinement checker, and several new tools for automatically building models of systems in CSP from existing artifacts, including source code. We have described the research that went into these tools and how they have been applied to the HACMS platforms. Many of the tools are now available as free, open-source software.

Penetration testing discovered numerous vulnerabilities in the original, unsecured platforms. These vulnerabilities guided the blue team performers, ensuring their work applied in practice. The final vehicles delivered under the HACMS program, even as research prototypes, proved to be resilient against most forms of attack to a degree rarely seen even in hardened, fielded systems. Of all the final, formally verified components assessed under the final phase of the program, no memory corruption failures, mathematical operation faults, or security isolation compromises were identified.

As highlighted in the companion Final Vehicle Security Assessment Report, there remain challenges in securing autonomous systems. First, the formal guarantees of security properties should be applied exhaustively throughout any system to be protected. Without careful application throughout all components integral to critical functionality, the system in question may remain highly vulnerable to broad classes of readily implemented cyber attack. Second, HACMS technologies should be further applied to other forms of system security properties. This is most clearly observable regarding communications security. For nearly every critical communications security weakness identified in the baseline systems, a corresponding critical vulnerability was found in the communications security of the newly implemented final HACMS vehicles. This is not to be taken as a weakness in the tools developed under HACMS, as none made the claim that they offered proven secure cryptographic protections. However, it does highlight the security gains possible by expanding the formally proven build processes developed under HACMS to include other security properties, communications security or otherwise.

Continuing to build on the successful application and advancement of HACMS techniques and technologies will continue to offer revolutionary advancements to the security of autonomous vehicles. At its hypothetical limit, the HACMS program has laid the groundwork for an autonomous vehicle for which there is no distinction between its functional specification and its operational behavior under duress; that is, a system for which an attacker provably cannot trigger anomalous behavior.

6. REFERENCES

[1] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[2] A. W. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, 1997.

[3] A. W. Roscoe, Understanding Concurrent Systems, Springer, 2010.

[4] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666-677, 1978.

[5] A. E. Abdallah, C. B. Jones and J. W. Sanders, Communicating Sequential Processes: The First 25 Years, Springer, 2004.

[6] T. Gibson-Robinson, P. Armstrong, A. Boulgakov and A. W. Roscoe, "Failures Divergences Refinement (FDR) Version 3," 2013. [Online]. Available: https://www.cs.ox.ac.uk/projects/fdr/.

[7] *ISO/IEC 9899:2011 -- Information technology -- Programming languages -- C*, Geneva, Switzerland: International Organization for Standardization, 2011.

[8] T. Gibson-Robinson, P. Armstrong, A. Boulgakov and A. W. Roscoe, "FDR3 - A Modern Refinement Checker for CSP," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2014.

[9] T. Gibson-Robinson and A. W. Roscoe, "FDR Into The Cloud," in *Communicating Process Architectures*, 2014.

[10] J. Lawrence, "Practical Application of CSP and FDR to Software Design," in *Communicating Sequential Processes: The First 25 Years*, 2005.

[11] A. Mota and A. Sampaio, "Model-checking CSP-Z: strategy, tool support and industrial application," *Science of Computer Programming*, vol. 40, no. 1, pp. 56-96, 2001.

[12] B. Shapiro and C. Casinghino, "specgen: A Tool for Modeling Statecharts in CSP," in *Nasa Formal Methods (NFM 2017)*, 2017.

[13] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231-274, 1987.

[14] E. Mikk, Y. Lakhnech, M. Siegel and G. J. Holzmann, "Implementing Statecharts in Promela/Spin," in *IEEE Workshop on Industrial Strength Formal Specification Techniques*, 1998.

[15] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin and J. D. Reese, "Model Checking Large Software Specifications," *IEEE Transactions on Software Engineering*, vol. 24, no. 7, pp. 498-520, 1998.

[16] A. W. Roscoe and Z. Wu, "Verifying Statemate Statecharts Using CSP and FDR," in *International Conference on Formal Engineering Methods*, 2006.

[17] B. Shapiro and C. Casinghino, "SpecGen," 2016. [Online]. Available: https://github.com/draperlaboratory/specgen.

[18] D. Harel and A. Naamad, "The Statemate Semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 4, pp. 293-333, 1996.

[19] R. Eshuis and R. Wieringa, "Requirements-Level Semantics for UML Statecharts," in *Formal Methods for Open Object-Based Distribution Systems*, 2000.

[20] C. Casinghino, "cspgen," 2016. [Online]. Available: https://github.com/draperlaboratory/cspgen.

[21] A. W. Appel, Compiling with Continuations, Cambridge University Press, 1992.

[22] T. C. d. team, "The Coq proof assistant reference manual," 2004. [Online]. Available: http://coq.inria.fr.

[23] B. C. Pierce, A. Azevedo de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg and B. Yorgey, "Software Foundations," 2009. [Online]. Available: https://www.cis.upenn.edu/~bcpierce/sf.

[24] G. Winskel, The Formal Semantics of Programming Languages: An Introduction, MIT Press, 1993.

[25] D. Laboratory, "Fracture," 2014. [Online]. Available: https://github.com/draperlaboratory/fracture.

[26] D. Brumley, I. Jager, T. Avgerinos and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *International Conference on Computer Aided Verification*, 2011.

[27] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[28] C. O'Halloran, "Verifying Critical Cyber-Physical Systems After Deployment," in *International Workshop on Automated Verification of Critical Systems*, 2015.

LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

AIS	Assured Information Security
CONOPS	Concept of Operations
CSP	Communicating Sequential Processes
DAG	Directed Acyclic Graph
DARPA	Defense Advanced Research Projects Agency
Draper	The Charles Stark Draper Laboratory
EA	Enterprise Architect
EC2	Elastic Compute Cloud
FDR	Failures Divergence Refinement
GLTS	Generalized Labelled Transition System
HACMS	High-Assurance Cyber Military Systems
IR	Intermediate Representation
JTAG	Joint Test Action Group
LLVM	Low Level Virtual Machine
Oxford	Oxford University
SMACCMPilot	Secure Mathematically-Assured Composition of Control Models Pilot
SSA	Single Static Assignment
TA	Technical Area
UART	Universal Asynchronous Receiver/Transmitter
UML	Unified Modeling Language